Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tilen Venko

Razvoj spletnih aplikacij, ki delujejo brez internetne povezave

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Viljan Mahnič

Ljubljana, 2017

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Razvoj spletnih aplikacij, ki delujejo brez internetne povezave

Tematika naloge:

Pri razvoju spletnih aplikacij se pogosto srečamo s problemom, kako zagotoviti njihovo delovanje tudi v primeru, ko uporabnik nima dostopa do internetne povezave. Enega od takih primerov predstavlja uporaba računalnikov v patronažni službi, kjer od patronažne sestre zahtevamo, da na terenu - med obiskom nekega pacienta - vnaša podatke o njegovem zdravstvenem stanju, meritvah krvnega tlaka, srčnega utripa, krvnega sladkorja ipd.

Proučite pristope, ki zagotavljajo čim bolj popolno funkcionalnost spletnih aplikacij tudi takrat, ko delujejo v nepovezanem načinu. Ocenite njihovo uporabnost in izberite tistega, ki je po vašem mnenju najboljši. Uporabo izbranega pristopa prikažite na zgoraj omenjenemu primeru iz patronažne službe in opišite ključne dele rešitve. Za predstavitev podatkov o pacientih in meritvah uporabite standard za izmenjavo zdravstvenih podatkov FHIR (Fast Healthcare Interoperability Resources).

Rad bi se zahvalil svojim staršem in prijateljem, ki so me podpirali pri mojem študiju in izdelavi diplomske naloge. Zahvala gre tudi podjetju Parsek d.o.o., predvsem gospodu Tomu Jarcu in gospodu Nenadu Živkoviću, ki sta mi velikodušno pomagala pri zasnovi in izdelavi spletne aplikacije. Posebna zahvala gre tudi mentorju prof. dr. Viljanu Mahniču za vso pomoč, vodenje in nasvete pri pisanju diplomske naloge.

Kazalo

Kazalo programske kode

Slike

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	Application Programming Interface	programski vmesnik
CSS	Cascading Style Sheets	jezik za vizualno oblikovanje strani
DOM	Document Object Model	objektni model dokumenta
FHIR	Fast Healthcare Interoperability Resources	hitra interoperabilnost virov v zdravstvu
HTML	HyperText Markup Language	označevalni jezik za pisanje spletnih strani
HTTPS	Hyper Text Transfer Protocol Secure	protokol, ki omogoča varno spletno povezavo
JSON	JavaScript Object Notation	objektna notacija JavaScript
SQL	Structured Query lLanguage	strukturiran povpraševalni jezik za delo s podatkovnimi bazami
URL	Uniform Resource Locator	spletni naslov

Povzetek

Naslov: Razvoj spletnih aplikacij, ki delujejo brez internetne povezave

Avtor: Tilen Venko

Ta diplomska naloga poskuša rešiti problem uporabe spletnih aplikacij, ko internetna povezava ni na voljo oziroma je zelo slaba. V njej smo skušali prikazati, kako ustvariti spletno aplikacijo, ki bo delovala in omogočala večino funkcionalnosti, tudi če uporabnik nima dostopa do internetne povezave. V nalogi najprej opišemo možne pristope k rešitvi te težave, opišemo njihove prednosti in slabosti, nato predstavimo delovanje in funkcionalnosti aplikacije, ter kako smo dosegli, da naša spletna aplikacija deluje tudi brez internetne povezave s pomočjo tehnologij ServiceWorker, IndexedDB in lokalno shrambo brskalnika.

Za domeno spletne aplikacije smo si izbrali zdravstveno informatiko oziroma natančneje aplikacijo, ki bo omogočala patronažnim sestram, da shranjujejo meritve pacientov in jih potem tudi pregledujejo.

Ključne besede: Spletna aplikacija, Service Worker, IndexedDB, Cache Storage, HTML5.

Abstract

Title: Developing offline enabled web applications

Author: Tilen Venko

In this thesis, we are trying to solve the problem of using the web applications when the user is offline or the internet connection is weak. We were trying to build a web application, which will be usable and will maintain the majority of functionalities even in offline mode. In the thesis, we first describe different possibilities and technologies, which enable us to solve this problem, explaining how they work and pointing out their pros and cons. Then we present our application, describe how it works, which functionalities it has, and how we've achieved that our web application can work in offline mode with the help of Service Worker, IndexedDB, and Local Storage.

We chose healthcare for the domain of our application, more precisely, we decided to build a web application for nurses, which will allow them to store patient observations and later view them.

Keywords: Offline Web Application, Service Worker, IndexedDB, Cache Storage, HTML5.

Poglavje 1

Uvod

Spletne aplikacije postajajo vse bolj popularne in praktično vse aplikacije se selijo na splet. Čeprav imajo spletne aplikacije v primerjavi s klasičnimi aplikacijami veliko prednosti, je njihova največja slabost ta, da za delovanje potrebujemo internetno povezavo. Kljub veliki pokritosti z internetom ali mobilnim omrežjem ta še vedno ni na voljo čisto povsod. Še vedno se pogosto zgodi, da ostanemo brez internetne povezave ravno takrat, ko jo najbolj potrebujemo. Lahko pa se zgodi tudi, da nam mobilna naprava prikazuje izvrsten signal, vendar se zalomi kje na poti od naše naprave do strežnika. Lahko ima ponudnik internetnih storitev težave, lahko je katero vozlišče na poti preobremenjeno in ne spušča prometa skozi, lahko se zgodi, da je naš strežnik preobremenjen ali pa ima programskega hrošča in je nehal delovati. Veliko je stvari, ki lahko pri internetni povezavi gredo narobe, mi pa želimo, da bi uporabnik lahko svoje delo opravljal neodvisno od tega, ali ima povezavo v svetovni splet ali ne.

Zato smo se odločili, da raziščemo tehnologije, ki to omogočajo, in naredimo aplikacijo, ki bo uporabna tudi, če uporabnik ne bo imel povezave v svetovni splet, oziroma bo v tako imenovanem nepovezanem načinu (ang. offline mode). Poleg tega, da s spletno aplikacijo, ki lahko deluje tudi v nepovezanem načinu, omogočimo delovanje vsepovsod in neodvisno od povezave v svetovni splet, ima nepovezan način tudi to prednost, da se aplikacija

naloži dosti hitreje, saj ni potrebno, da zahtevano spletno aplikacijo prenesemo s strežnika, ampak jo naložimo iz lokalne shrambe brskalnika, kar je veliko hitreje. S tem omogočimo boljšo uporabniško izkušnjo ter povečamo zanesljivost in robustnost spletne aplikacije.

Zdravstvo je še eno redkih področji, ki ni digitalizirano. Ena od prednosti, ki bi jih prineslo digitalizirano zdravstvo, je tudi to, da bolnikom ne bi bilo treba hoditi v bolnišnice in zdravstvene domove, ampak bi bili oskrbljeni na svojem domu. V aplikacijah, ki bi podpirale digitalizacijo procesov v zdravstvu, bi poleg varnosti tako morali poskrbeti tudi za zanesljivost delovanja aplikacij na terenu. Zaradi razlogov, ki smo jih našteli v prejšnjem odstavku, vemo, da ne moremo pričakovati, da bi na terenu vedno imeli povezavo v svetovni splet oziroma, da bi ta bila zanesljiva. Ker se nam zdi področje digitalizacije zdravstva zanimivo, hkrati pa bi rešili realen problem uporabnosti spletnih aplikacij na terenu, smo se odločili, da naredimo prototip spletne aplikacije za patronažne sestre, ki bo uporabna tudi v nepovezanem načinu. Spletna aplikacija za patronažne sestre bo omogočala osnovno planiranje obiskov in zajemanje ter pregled meritev na obiskih.

V nadaljnjem besedilu bomo najprej v drugem poglavju opisali vse tehnologije, ki vsaj delno omogočajo izgradnjo nepovezanih (ang. offline) aplikacij, razložili, kako delujejo in izpostavili njihove prednosti in pomanjkljivosti. V tretjem poglavju bomo opisali vse ostale tehnologije in standarde, ki so bili potrebni za razvoj spletne aplikacije. Nato pa se bomo v četrtem poglavju osredotočili na razvoj spletne aplikacije za patronažne sestre, predstavili, kaj vse omogoča in razložili, kako smo dosegli, da deluje brez internetne povezave. Na koncu pa bomo v petem poglavju povzeli še sklepne ugotovitve.

Poglavje 2

Predstavitev tehnologij, ki omogočajo delovanje spletnih aplikacij brez interneta

S prihodom tehnologije HTML5 smo dobili prva resna orodja, ki omogočajo izgradnjo nepovezanih spletnih aplikacij. Pred tem so bili edini standardiziran način za shranjevanje podatkov piškotki, s pomočjo katerih pa ni mogoče zgraditi spletne aplikacije, ki bi delovala v nepovezanem načinu, ampak samo shranjujejo podatke o seji. Novejše tehnologije pa nam omogočajo, da lahko na odjemalčevi strani implementiramo podatkovno bazo, shranjujemo podatke in spletne vire v lokalno shrambo, prestrezamo zahteve uporabnika in odgovore strežnika ter glede na status odgovora primerno odreagiramo.

Namen tega poglavje je opisati tehnologije, ki nam omogočajo izdelavo nepovezanih spletnih aplikacij, predstaviti, kakšen je njihov namen, katere težave rešujejo in izpostaviti morebitne pomanjkljivosti.

2.1 HTML5 shramba

HTML5 shramba (ang. HTML5 Storage) je najbolj preprosta lokalna shramba, ki je bila definirana v tehnologiji HTML5 in je prva ponujala nekaj mega-

bajtov prostora, ki so ga razvijalci lahko uporabili, da so lokalno shranjevali podatke. Pred pojavom HTML5 shrambe ni obstajal standard, ki bi definiral, kako se podatki lokalno shranjujejo v brskalnikih, ampak je to vsak brskalnik delal po svoje, obstajala pa je tudi vrsta vtičnikov, ki pa niso bili podprti s strani vseh brskalnikov, ali pa so imeli omejene funkcionalnosti. Čeprav je HTML5 shramba zelo omejena, je predstavljala velik napredek, saj je to prvi standard, ki so se ga držali vsi brskalniki, prav tako pa je v primerjavi s prejšnjimi rešitvami ponujala veliko več prostora, nekaj megabajtov v primerjavi z nekaj kilobajti [19, 8, 7].

Vse, kar nam HTML5 shramba ponuja, je shranjevanje parov ključ, vrednost, pri čemer so vrednosti vedno shranjene kot nizi (ang. string). Pri shranjevanju drugih tipov, kot so številski ali logični, moramo biti zato previdni, da jih ob pridobitvi iz shrambe pretvorimo spet v prvotne tipe. HTML5 shrambo razširjata dva tipa shrambe:

- **LocalStorage** - podatki so trajno shranjeni in ostanejo tudi, ko zapremo zavihek, brskalnik ali ugasnemo računalnik. Podatki ostanejo shranjeni, dokler jih eksplicitno ne odstranimo.
- **SessionStorage** - podatki, ki so shranjeni tukaj so vezani na sejo, kar pomeni, da se ob zaprtju zavihka ali okna podatki izgubijo, saj se s tem zapre tudi naša seja.

Čeprav je HTML5 shramba bila prvi standard in je še vedno podprta v vseh brskalnikih, je njena uporabnost zelo omejena in je primerna samo za enostavne aplikacije. Omogoča nam namreč premalo nadzora in ponuja premalo funkcionalnosti.

2.2 Application Cache

Programski vmesnik Application Cache je prva rešitev, ki se je pojavila s prihodom tehnologije HTML5 in je omogočala vzpostavitev lokalne shrambe, v kateri so shranjene spletne strani in drugi spletni viri, kot so slike ali skripte,

do katerih lahko dostopamo v nepovezanem načinu. Omogoča nam večji nadzor nad upravljanjem shranjenih spletnih virov v lokalni shrambi in več funkcionalnosti kot HTML5 shramba, ki je omogočala samo shranjevanje podatkov v obliki parov ključ-vrednost [9, 12, 7].

Spletne strani lahko razdelimo na dve kategoriji: strani, na katerih vsebino dobimo oziroma jo iščemo in strani, na katerih lahko vsebino ustvarjamo.

- Strani, na katerih iščemo, so npr. Wikipedia, Facebook, spletna stran lokalne kavarne. Na takšnih straneh je glavno delo opravljeno na strežnikih, brskalnik na odjemalcu pa uporabniku samo prikaže stran.

- Na straneh, na katerih lahko ustvarjamo vsebino, kot so Google Docs, Office 365 ali spletne igre, pa je obremenitev na odjemalcu veliko večja. Te strani ponujajo omejeno količino podatkov, ki jih lahko uporabljamo na različne načine oz. kreiramo svojo vsebino in podatke. In ravno za te strani je Application Cache primeren.

2.2.1 Manifest

Če želimo razumeti, kako deluje Application Cache, si moramo najprej ogledati, kaj je to manifest dokument. Manifest dokument je preprost tekstovni dokument, ki pove brskalniku, katere HTML strani in druge dokumente mora shraniti v lokalno shrambo, da bo lahko do njih dostopala tudi v nepovezanem načinu [9, 5, 12, 7].

Manifest dokument je razdeljen na tri glavna področja:

- CACHE: tu navedemo vse HTML strani, CSS dokumente, JavaScript dokument, slike in ostale dokumente, ki jih želimo shraniti v lokalno shrambo brskalnika.

- NETWORK: datotekam, ki so tukaj navedene, dovolimo, da so prenesene s spleta. S posebno oznako, zvezdico (*), označimo, da to dovolimo

vsem dokumentom. Če kateri dokument pozabimo vključiti, tega ne bo mogoče pridobiti s strežnika, saj nam manifest dokument to preprečuje.

- FALLBACK: če želimo pridobiti stran, ki je nimamo shranjene v lokalni shrambi in smo v nepovezanem načinu, bodo postreženi dokumenti, ki smo jih shranili pod sekcijo fallback.

V kodi 2.1, je predstavljen primer manifest dokumenta, kjer v sekciji *CACHE* naveden html dokument *index.html*, stilska datoteka *stylesheet.png*, slika *slike/logo.png* in skripta *skripte/main.js*, ki jih želimo shraniti v lokalno shrambo. V sekciji *NETWORK*, dovolimo vsem datotekam, da so prenesene s spleta, v sekciji *FALLBACK*, pa sta navedena html datoteka *static.html* in slika *slike/offline.jpg*, ki se postrežeta v primeru napake. Vrstice označene z lojtro (#), predstavljajo komentarje. Komentarji so zelo koristi tudi zato, ker kot bomo izvedeli v poglavju 2.2.4, se preveri, če so dokumenti bili posodobljeni samo, če se je tudi sam manifest dokument posodobil. To lahko preprosto dosežemo s spreminjanjem verzije manifest dokumenta v komentarju.

```
1  CACHE MANIFEST
2  # v1-manifest-dokument
3
4  # Dokumenti, ki se bodo shranili v lokalno shrambo
5  CACHE:
6  index.html
7  stylesheet.css
8  slike/logo.png
9  skripte/main.js
10
11 # Dokumenti, katerim dovolimo, da jih uporabnik pridobi s
       spleta
12 # * pomeni vse dokumente
13 NETWORK:
14 *
15
16 # static.html bo postrezen, ce /main.py ni dostopen
```

```
17 # offline.jpg bo postrezena na vseh mestih, kjer dostopamo do
     slike/velike
18 FALLBACK:
19  static.html
20  slike/offline.jpg
```

Programska koda 2.1: Primer manifest dokumenta

2.2.2 Vzpostavitev lokalne shrambe

Da omogočimo shranjevanje v lokalno shrambo, moramo na vsako stran, ki jo želimo imeti lokalno shranjeno, dodati manifest atribut v HTML značko (kot je to prikazano v izseku programske kode 2.2) in jo dodati v seznam v samem manifest dokumentu pod sekcijo CACHE. Brskalnik potem avtomatsko doda vsako tako stran v lokalno shrambo.

```
1 <html manifest="primer.appcache">
2   ...
3 </html>
```

Programska koda 2.2: Manifest atribut v HTML znački

2.2.3 Nalaganje dokumentov

Uporaba programskega vmesnika Application Cache, spremeni normalno delovanje nalaganja spletnih strani, ki sedaj poteka tako [12]:

1. Če je spletna stran shranjena v lokalni shrambi in je programski vmesnik Application Cache že vzpostavljen, brskalnik stran naloži takoj iz svojega pomnilnika, brez dostopanja do interneta. Če pa se stran še ne nahaja v lokalni shrambi, Application Cache spletno stran prenese s strežnika in vse strani, ki so navedene v manifest dokumentu, doda v lokalno shrambo. Od sedaj naprej bo brskalnik vedno naložil to stran iz lokalne shrambe.

2. Ko je stran naložena, brskalnik preveri, ali je bila stran na strežniku spremenjena.

3. Če Application Cache ugotovi, da se je vsebina strani spremenila, vse dokumente iz lokalne shrambe premesti v začasno shrambo, iz strežnika pridobi posodobljeno verzijo strani in če pridobi vse strani brez napake, doda strani v lokalno shrambo, stare strani pa izbriše.

4. Ker je stran že naložena, je kljub temu, da ima novo vsebino, brskalnik ne prikaže takoj, ampak počaka, da uporabnik osveži stran.

5. Če se internetna povezava prekine, bomo lahko še vedno nemoteno uporabljali stran, dokler dostopamo samo do strani, ki jih imamo shranjene v lokalni shrambi. Ko pa želimo dostopati do strani, ki je prej nismo shranili, se nam bo prikazala vsebina, ki je navedena v *FALLBACK* sekciji manifest dokumenta.

6. Ko se katera od spletnih strani na strežniku spremeni, bi pričakovali, da ob osvežitvi strani dobimo najnovejše podatke. Vendar ni tako, saj se stran vedno naloži iz lokalne shrambe, nato pa se preveri samo, če se je posodobil manifest dokument. Zato moramo biti pazljivi, da vedno, ko posodobimo kakšno od strani, posodobimo tudi manifest dokument, drugače se vsebina pri uporabniku ne bo nikoli posodobila.

7. Torej šele, ko posodobimo tako spletno stran kot manifest dokument, bo brskalnik zaznal spremembe in posodobil podatke v lokalni shrambi. Vendar se nam tudi tokrat ob osvežitvi strani še ne prikaže najnovejša stran, saj kot smo omenili zgoraj, brskalnik najprej naloži stran, šele nato preveri morebitne posodobitve.

2.2.4 Pomanjkljivosti vmesnika Application Cache

Ker brskalnik vedno najprej naloži vsebino iz lokalne shrambe in nato preveri, ali se je na strežniku vsebina strani spremenila, lahko nastane težava. Če imamo v svojem manifest dokumentu navedenih veliko število HTML strani, bo brskalnik moral ob vsakem dostopu na eno od teh strani preveriti za vse strani, ali so se posodobile, kar pa pomeni veliko prometa. Da bi se temu

izognili, brskalniki pravzaprav preverjajo, ali se je vsebina strani spremenila samo, če se je spremenila sama vsebina manifest dokumenta.

Naslednji problem se pojavi, ker imajo lahko strani v lokalni shrambi značke nikoli me ne shrani, vedno preveri na strežniku, če je na voljo posodobitev, ali pa privzemi, da sem veljavna do določenega datuma. Tako se lahko zgodi, da HTML stran označimo z veljavnim datumom in je potem nikakor ne moremo posodobiti, čeprav smo jo posodobili na strežniku in prav tako tudi manifest dokument.

Še večjo napako lahko naredimo, če samemu manifest dokumentu dodelimo datum, do katerega je veljaven, ker se potem ne bo nikoli nič posodobilo, dokler datum veljavnosti na manifest dokumentu ne poteče, saj kot smo omenili zgoraj, brskalnik preverja ali so se posodobile HTML strani, samo če se je posodobil manifest dokument, kar pa se v tem primeru ne more zgoditi.

Problem se lahko pojavi tudi, če pozabimo dodati kakšen vir (spletno stran, sliko, skripto...) v manifest dokument, saj ta ne bo prikazan, tudi če imamo internetno povezavo. Problem se sicer da rešiti tako, da spremenimo razdelek NETWORK v manifest dokumentu [2].

Glavne pomanjkljivosti vmesnika Application Cache, so torej, da ne prikaže najnovejše vsebine, čeprav imamo povezavo na splet, razvijalcem omogoča premalo kontrole nad spletnimi viri in podatki, ki so lokalno shranjeni, in ne omogoča, da ko prvič obiščemo spletno stran, dobimo vse HTML strani, ki jih potrebujemo takrat, ko internetne povezave ni. Zaradi vseh naštetih problemov Application Cache API ni najbolj priljubljen. Prav tako ga ustvarjalci spletnih brskalnikov odsvetujejo, saj je označen, kot opuščena (ang. deprecated) tehnologija.

2.3 WebSQL

WebSQL je programski vmesnik, ki temelji na podatkovni bazi SQLite in ni del standarda HTML5. Kot tak, ni nikoli prav zaživel, saj so ga kmalu nehali

razvijati zaradi problema standardizacije. WebSQL ni bil nikoli podprta v vseh brskalnikih, saj so ga podprli samo brskalniki Chrome, Opera in Safari. Tehnologija pa poleg tega, da ni podprta v vseh brskalnikih ni priporočljivo uporabljati predvsem zato, ker je označena kot opuščena (ang. deprecated), kar pomeni, da je razvijalci zanjo ne nudijo več podpore in je ne razvijajo več.
WebSQL nam omogoča, da lahko na odjemalčevi strani uporabljamo SQL, kar pomeni, da imamo tudi na strani odjemalca podatkovno bazo, v katero trajno shranjujemo podatke, nad njimi izvajamo poizvedbe in uporabljamo transakcije. Ker tehnologija ni nikoli doživela velike podpore in so jo hitro označili, kot opuščeno, je ne bomo podrobneje opisovali [20, 11].

2.4 Service Worker

Service Worker je skripta, ki jo brskalnik izvaja v ozadju, neodvisno od spletne strani, ki jo streže. Njena glavna naloga in prednost je, da lahko prestreže zahtevke odjemalca in odgovore strežnika ter glede na njihov status primerno odreagira.

Service Worker je JavaScript Worker [4], kar pomeni, da ne more neposredno dostopati in spreminjati zgradbe spletne strani (ang. Document Object Model, DOM), ampak lahko manipulira z vsebino dokumenta posredno preko vmesnika *postMessage* [6]. Ko brskalnik zazna, da spletna stran, ki streže skripto Service Worker, ni več odprta v nobenem zavihku ali oknu in s tem tudi skripta ni več v uporabi, jo ugasne, ko pa je spet potrebna zažene novo instanco le-te. Zaradi tega vanjo ne moremo shranjevati trajnih podatkov, ampak za to raje uporabimo tehnologijo IndexedDB, ki jo bomo opisali v naslednjem poglavju 2.5. Service Worker nam ponuja veliko poslušalcev (ang. event listeners), s katerimi lahko zajamemo veliko možnih scenarijev v nepovezanem načinu [10, 3].

2.4.1 Življenjski cikel skripte Service Worker

Kot smo že omenili, je življenjski cikel skripte Service Worker povsem ločen od spletne strani. Najprej jo je potrebno registrirati, kar storimo v JavaScript skripti naše spletne strani. Registracija bo sprožila namestitev skripte Service Worker v ozadju. Ob namestitvi po navadi določimo, katere vire naj shrani v lokalno shrambo. Če namestitev uspe, vemo, da so naši viri sedaj na voljo tudi v nepovezanem načinu, po namestitvi pa se sproži postopek aktivacije. Ob aktivaciji lahko poskrbimo za stare verzije skripte Service Worker in lokalne shrambe, če je to potrebno in ta obstaja. Ko pa se aktivacija konča, Service Worker prevzame nadzor nad spletno stranjo. Če pa registracija oziroma namestitev ni uspešna, seveda naši viri ne bodo shranjeni v lokalni shrambi, bo pa ob naslednji osvežitvi strani brskalnik ponovno poskusil namestiti skripto Service Worker.

Ko do spletne strani dostopamo prvič, Service Worker še ne prevzame kontrole nad spletno stranjo, saj nalaganje spletne strani poteka tako, da brskalnik s strežnika pridobi osnovni HTML dokument, začne nalagati ta dokument in hkrati zahteva od strežnika še vse ostale vire, ki so navedeni s povezavami v HTML dokumentu, kot so slike, stilske datoteke css in skripte, med njimi tudi skripto Service Worker. Ker je brskalnik naložil spletno stran še preden je skripta Service Worker obstajala, ta ne prevzame nadzora nad spletno stranjo, ko pa spletno stran osvežimo, skripta prevzame nadzor, saj ta obstaja še preden se je stran naložila.

Sedaj, ko razumemo, kako se skripta Service Worker naloži ob prvem dostopu do spletne strani, si oglejmo, kako jo lahko posodabljamo. Recimo, da smo posodobili skripto, in ko bo uporabnik naslednjič osvežil stran, bo brskalnik v ozadju preveril, ali se je skripta posodobila. Ker se ta je posodobila, jo prenese in postane nova verzija skripte Service Worker, vendar še ne prevzame nadzora nad spletno stranjo, temveč čaka. Nova verzija skripte ne bo prevzela nadzora nad spletno stranjo, dokler niso vse instance te strani, ki so lahko odprte v več zavihkih ali oknih, zaprte in ne uporabljajo več stare verzije skripte. To zagotavlja, da vse instance spletne strani uporabljajo isto

verzijo skripte in s tem zagotovimo konsistentnost.

Logično bi iz tega sledilo, da če ima uporabnik odprto samo eno instanco spletne strani, se bo ob osvežitvi spletne strani namestila nova verzija skripte in prevzela nadzor nad spletno stranjo. Vendar ni tako, razlog pa je brskalnikov način posodabljanja spletnih strani. Namreč, ko osvežimo spletno stran, na enkrat obstajata dve instanci spletne strani, stara in nova. Staro instanco nadzira še stara verzija skripte, zato nova verzija ne more prevzeti nadzora. Ne glede na to, kolikokrat osvežimo spletno stran, nova verzija skripte ne bo prevzela nadzora, dokler popolnoma ne zapremo spletne strani in jo ponovno odpremo v novem zavihku ali oknu brskalnika. To obnašanje je lahko moteče, vendar, če želimo zagotoviti konsistentnost, ni druge izbire.

Kljub temu, da se skripta ne more avtomatsko posodobiti, dokler strani popolnoma ne zapremo, lahko uporabnika obvestimo o novi verziji skripte in mu, na primer s klikom na gumb *posodobi*, omogočimo, da posodobi skripto brez potrebe po zapiranju strani. To nam omogoča metoda *skipWaiting()*, ki obide običajni življenski cikel skripte in omogoči, da čakajoča nova verzija skripte takoj prevzame nadzor nad spletno stranjo.

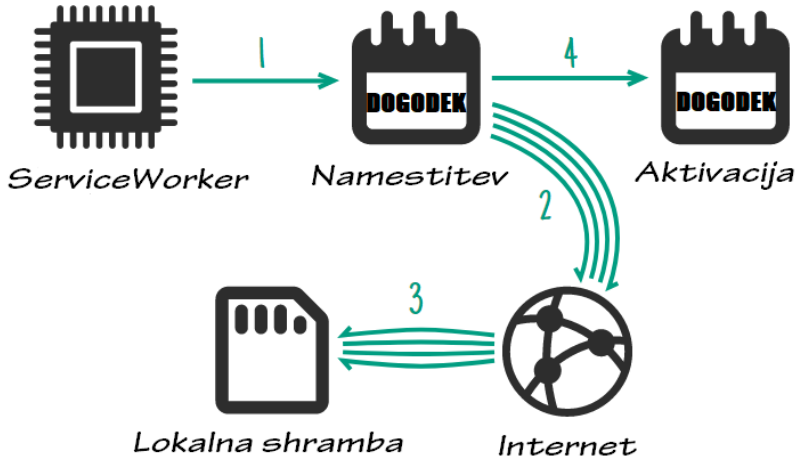
2.4.2 Kako in kdaj shraniti podatke v lokalno shrambo

Podatke v lokalno shrambo načeloma shranjujemo ob namestitvi in aktivaciji skripte Service Worker, interakciji uporabnika s spletno stranjo, ob odgovoru strežnika ali pa ob sinhroniziranju v ozadju. V nadaljevanju bomo opisali naštete primere in obrazložili, kateri podatki se tipično shranjujejo pri danih primerih.

- **Ob namestitvi** - Kot lahko vidimo v programski kodi 2.3 primera namestitve skripte Service Worker, ta pozna dogodek *install*, ki se sproži ob namestitvi nove skripte in se zgodi pred vsemi drugimi dogodki. Tukaj si pripravimo stvari, ki jih želimo shraniti v lokalno shrambo, predvsem statične datoteke, ki so potrebne, da naša spletna stran sploh deluje. Pri namestitvi lahko ločimo dve skupini virov, tiste, brez katerih naša spletna stran ne more delovati in tiste, ki niso tako ključni, ali

pa jih bomo potrebovali šele pozneje in jih bomo morda lahko pridobili takrat. Na tiste vire, ki so ključni, moramo ob namestitvi počakati. To storimo z funkcijo *event.waitUntil*, ki čaka z namestitvijo, dokler ne pridobimo vseh ključnih virov in jih uspešno shranimo v lokalno shrambo. Če ta postopek ne uspe, tudi namestitev ni uspešna. Viri, ki niso ključni, se razlikujejo po tem, da funkcija *event.waitUntil* ne čaka odgovora na njihovo uspešno pridobitev. Slika 2.1 prikazuje potek namestitve skripte.

Pomembno si je zapomniti, da če obstaja stara verzija skripte, v tej fazi ta še vedno teče, tako da v kodi za namestitev ne smemo narediti ničesar takega, kar bi motilo oz. onemogočilo delovanje prejšnje verzije.



Slika 2.1: Ob namestitvi (povzeto po [3])

```
self.addEventListener('install', function(event) {
   event.waitUntil(
     caches.open('lokalna-shramba-v1').then(function(cache)
   {
        cache.addAll(
          // viri, brez katerih bi nasa spletna stran tudi
   delovala
          '/slike/primer_slike.png'
```

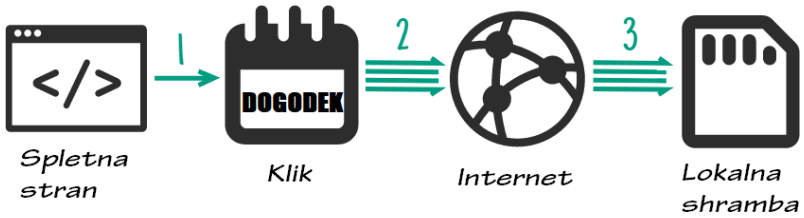
```
7           );
8           return cache.addAll(
9            // viri, ki so kljucni za naso spletno stran
10           'index.html',
11            '/css/glavni.css',
12            '/js/glavni.js'
13          );
14      })
15  );
16 });
```

Programska koda 2.3: Service Worker - dogodek ob namestitvi (povzeto po [3])

- **Ob aktivaciji** - Ko je skripta uspešno nameščena, starejša verzija pa odstranjena, se sproži dogodek ob aktivaciji. Koda, ki jo izvajamo med aktivacijo, naj bo čim krajša, saj so takrat vsi ostali dogodki postavljeni v vrsto, kar pomeni, da se spletna stran ne bo naložila, dokler se koda v aktivaciji ne izvede do konca. Tako je edina smiselna koda, ki se izvaja tukaj ta, da pobrišemo neuporabljene podatke v lokalni shrambi in posodobimo podatkovno bazo IndexedDB, torej to, česar ne moremo narediti, dokler še teče stara verzija skripte Service Worker. Na sliki 2.2 je prikazan potek aktivacije skripte.

Slika 2.2: Ob aktivaciji (povzeto po [3])

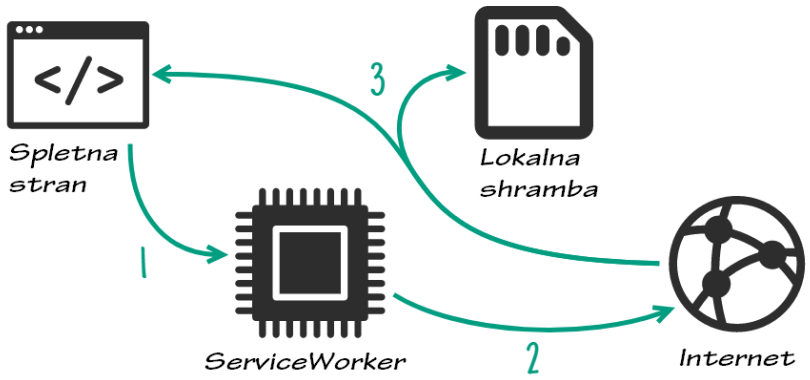
- **Ob interakciji uporabnika** - Ta poslušalec je zelo uporaben, saj lahko uporabniku ponudimo možnost, da izbere vsebino, ki jo želi shraniti v lokalno shrambo, da bo do nje lahko dostopal v nepovezanem načinu. Tukaj je bolj kot shranjevanje celotnih strani, primerno za shranjevanje videov, slikovne galerije oziroma ostalih virov, ki so del spletne strani in se naložijo ločeno, uporabnik pa bi želel do njih dostopati tudi v nepovezanem načinu. Slika 2.3 prikazuje potek shranjevanja virov v lokalno shrambo na zahtevo uporabnika.



Slika 2.3: Ob interakciji uporabnika (povzeto po [3])

- **Ob odgovoru strežnika** - Ko imamo spletno aplikacijo naloženo, je po navadi potrebno vsebino spletne aplikacije tudi redno posodabljati, saj spletne aplikacije niso namenjene statični rabi, temveč interakciji z uporabnikom. Tako bo spletna aplikacija enkrat, ko je že naložena zahtevala novo vsebino, ki jo je potrebno posodobiti ali na novo naložiti. Kot lahko vidimo v programski kodi 2.4, bo skripta v tem primeru najprej poskušala pridobiti vir iz lokalne shrambe, če to ne bo mogoče, pa bo zahtevala ta vir od strežnika. Ob odgovoru strežnika bo skripta hkrati posodobila vsebino spletne aplikacije in shranila vir v lokalno shrambo.

  Pri tem je potrebno paziti, da lokalne shrambe preveč ne obremenimo z nepotrebnimi viri, saj se lahko zgodi, da bo brskalnik v primeru pomanjkanja pomnilnika primoran pobrisati podatke, mi pa ne želimo, da bi pobrisal naše lokalno shranjene podatke, ker jih imamo preveč. Zato je zelo pomembno, da vire, ki jih ne potrebujemo več, sproti odstranjujemo. Na sliki 2.4 je prikazan potek shranjevanja v lokalno shrambo ob odgovoru strežnika.



Slika 2.4: Ob odgovoru strežnika (povzeto po [3])

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open('dinamicna-shramba').then(function(cache) {
```

```
4       return cache.match(event.request).then(function (
    response) {
5         // Vrnemo vir iz lokalne shrambe ali pa s streznika
6         return response || fetch(event.request).then(
    function(response) {
7           // Ce dobimo nove vire s streznika, jih hkrati
    tudi shranimo v lokalno shrambo
8           cache.put(event.request, response.clone());
9           return response;
10        });
11      });
12    })
13  );
14 });
```

Programska koda 2.4: Service Worker - ob odgovoru strežnika (povzeto po [3])

- **Sinhronizacija v ozadju** - Je primerna, ko želimo sinhronizirati podatke oziroma uporabnika obvestiti o novem obvestilu, tudi, če nima odprte spletne aplikacije v brskalniku. Seveda mora biti uporabnik v času prejema obvestila oziroma posodabljanja povezan v internet, vendar pa lahko do vsebine obvestila oziroma podatkov, ki so se prenesli v ozadju, dostopa tudi, ko internetna povezava ni na voljo.

2.4.3 Prestrezanje in odgovarjanje na zahteve

Skripta Service Worker za razliko od programskega vmesnika Application Cache ne streže virov avtomatsko iz lokalne shrambe, ampak ji moramo eksplicitno povedati, kdaj in katere vire naj streže iz lokalne shrambe. Za to obstaja nekaj uporabnih metod:

- **Samo lokalna shramba** - Skripta Service Worker bo preverila, ali vir obstaja v lokalni shrambi. Če ta ne obstaja, bo uporabnik dobil napako, kot v primeru, ko želi dostopati do spletne strani, vendar ta ni

na voljo. Čeprav imamo možnost, da Service Worker poskuša pridobiti vsebino samo iz lokalne shrambe, tega ni priporočljivo uporabljati, saj s tem ne pridobimo ničesar, lahko pa se zalomi pri pridobivanju vira.

- **Samo internet** - Skripta Service Worker bo poskušala vire pridobiti samo s strežnika, ne da bi prej preverila, ali so morda ti shranjeni v lokalni shrambi. Ta način je primeren predvsem za zahteve, ki nimajo nepovezanih (ang. offline) alternativ, kot so zahteve na strežnik, ki pridobivajo meta podatke ali pa pošiljanje podatkov na strežnik.

- **V primeru napake lokalne shrambe, uporabi internet** - Zelo primeren način, s katerim pridobimo večjo odzivnost in hitrost, saj je pridobivanje virov iz lokalne shrambe hitrejše kot pridobivanje virov s strežnika, hkrati pa v primeru, da vira nimamo shranjenega v lokalni shrambi, tega še vedno lahko pridobimo s strežnika. Ta primer pokrije oba zgoraj našteta primera in je zato veliko bolj primeren in preprost za uporabo.

- **V primeru napake interneta uporabi lokalno shrambo** - Primerno za uporabo pri virih, ki se hitro spreminjajo, kot so recimo članki na spletni strani. S tem dosežemo, da uporabnik vedno dobi najnovejše podatke, hkrati pa v primeru, da internetne povezave ni na voljo še vedno lahko dostopa do starejše vsebine, ki jo imamo lokalno shranjeno.

- **Tekma med internetom in lokalno shrambo** - Hkrati dostopamo do lokalne shrambe in strežnika ter vrnemo vir, ki ga dobimo najhitreje.

- **Najprej lokalna shramba in nato internat** - V tem primeru skripta Service Worker najprej naloži vire z lokalne shrambe, hkrati pa jih prenese tudi s strežnika, saj se je vsebina morda posodobila. Na novo pridobljene vire nato shrani v lokalno shrambo, da bodo naslednjič na voljo novejši podatki. Ni pa priporočljivo, da bi spletno stran osvežili

takoj, ko dobimo podatke s strežnika, saj je to lahko moteče za uporabnika.

- **Generična napaka**- Če vira ni mogoče pridobiti iz lokalne shrambe, ker si ga nismo prej shranili, in prav tako ne s strežnika, saj smo v nepovezanem načinu, lahko uporabniku vrnemo spletno stran, ki ga bo obvestila o napaki. Prednost je ta, da uporabnik dobi sporočilo o napaki, ki smo ga sestavili mi, namesto privzetega sporočila o napaki brskalnika.

2.4.4 Varnost

Ker je Service Worker zelo močno orodje, se nam upravičeno postavlja vprašanje varnosti. Brskalniki poskrbijo za varnost tako, da se vsi procesi skripte Service Worker zaganjajo v peskovniku, kar onemogoči nalaganje škodljive, kode na uporabnikov računalnik. Service Worker lahko registriramo samo, če imamo varno povezavo (uporabljamo protokol HTTPS), zanje pa velja tudi politika istega izvora (ang. same-origin policy), kar pomeni, da lahko do skripte Service Worker dostopajo samo strani z istim URL naslovom oziroma podnaslovom, s katerega je bila skripta registrirana.

Zavedati se moramo, da lokalna shramba ni trajna shramba in lahko ob pomanjkanju prostora v pomnilniku brskalnik tudi izbriše tiste podatke, za katere meni, da so odvečni. Seveda pa ne zna ločiti vsebine podatkov in vedeti, kateri podatki so za nas res pomembni in kateri so tisti, brez katerih bo naša aplikacije še vedno funkcionalna. Za premostitev te težave lahko uporabimo programski vmesnik *requestPersistent*, ki omogoči, da brskalnik ključne vire obravnava drugače in jih ne izbriše v primeru pomanjkanja prostora.

2.5 IndexedDB

IndexedDB je programski vmesnik (ang. API) za strukturirano shranjevanje podatkov v podatkovno bazo na strani odjemalca. Podobno kot HTML5 shramba tudi IndexedDB za shranjevanje uporablja povezavo med ključem in vrednostjo, vendar za razliko od HTML5 shrambe omogoča shranjevanje večje količine podatkov, ki so dobro strukturirani, podpira pa tudi transakcije [12, 14, 16, 7].

2.5.1 Osnovni koncepti podatkovne baze IndexedDB

- **IndexedDB shranjuje podatke v obliki ključ - vrednost.** Vrednost je lahko preprostega tipa (niz, število ...) ali pa strukturiran JavaScript objekt, ključi pa lahko predstavljajo lastnost tega objekta (recimo priimek pri uporabniku), lahko so avtomatsko generirane zaporedna števila, ali pa so katerikoli druga poljubna vrednost, ki jo določimo. Ključi služijo za indeksno iskanje po podatkovni bazi in ni potrebno, da so unikatni.

- **IndexedDB podpira transakcije**. Vse operacije nad podatkovno bazo, dodajanje novega objekta in iskanje po podatkovni bazi se izvajajo v transakcijah, kar pomeni, da se vsaka interakcija izvede v celoti ali pa sploh ne. Transakcije se ob uspešni izvedbi potrdijo same in jih ni moč ročno potrditi.

- **Operacije nas IndexedDB so asinhrone.** Namesto da bi se funkcije izvajale sinhrono in bi ob operaciji nemudoma dobili rezultat oziroma bi nanj čakali, dokler ga ne bi dobili, podatkovni bazi IndexedDB podamo povratno funkcijo, preko katere nas bo obvestila o uspešnosti operacije in vrnila morebitne podatke, ko bo operacija končana. Če nad podatkovno bazo izvajamo poizvedbe, nam ta ne vrne vrednosti, ampak ji moramo podati povratno funkcijo (ang. callback function), preko katere bomo dobili iskane podatke.

- **IndexedDB je objektno orientirana podatkovna baza.** Podatki so shranjeni v JavaScript objektih, katerih strukturo in ključe, po katerih bomo lahko iskali, določimo ob inicializaciji podatkovne baze. To shrambo imenujemo preprosto objektna shramba. Vsak objekt ima lahko več ključev oziroma indeksov, po katerih lahko poizvedujemo.

- **IndexedDB je omejena na izvor**, kar pomeni, da je dostopna samo iz izvirnega naslova. Ta omejitev velja zaradi varnostnih razlogov, da ne more neka druga spletna stran oziroma aplikacija dostopati do podatkov, shranjenih v naši podatkovni bazi IndexedDB.

2.5.2 Primerjava IndexedDB s relacijskimi bazami in HTML5 shrambo

V nasprotju z relacijskimi bazami podatki v podatkovni bazi IndexedDB niso shranjeni v naprej določenih tabelah s stolpci in vrsticami, temveč so shranjeni kot objekti v shrambah (ang. store), med katerimi ni relacij. Zaradi tega po podatkih v podatkovni bazi IndexedDB ne moremo poizvedovati s SQL stavki, ampak od nas zahteva, da podatke iz različnih shramb v programski kodi sestavljamo sami. Prav tako kot relacijske baze pa IndexedDB podpira transakcije.

Podobno kot v HTML5 shrambi so tudi tukaj podatki shranjeni s povezavo ključ vrednost, vendar IndexedDB ponuja več funkcionalnosti in bolje organizirane podatke. IndexedDB omogoča vračanje ključev po urejenem redu, preverjanje duplikatov, učinkovito iskanje na podlagi ključev, itd.

Poglavje 3

Predstavitev tehnologij in ogrodij, potrebnih za razvoj spletne aplikacije

3.1 HTML5, CSS3, JavaScript, TypeScript

HTML, CSS in JavaScript oziroma v našem primeru TypeScript, so osnovni programski jeziki potrebni za razvoj spletne strani ali spletne aplikacije. V tem odstavku bomo na kratko opisali vsakega od njih, da bo bralec lažje sledil razlagi delovanja aplikacije.

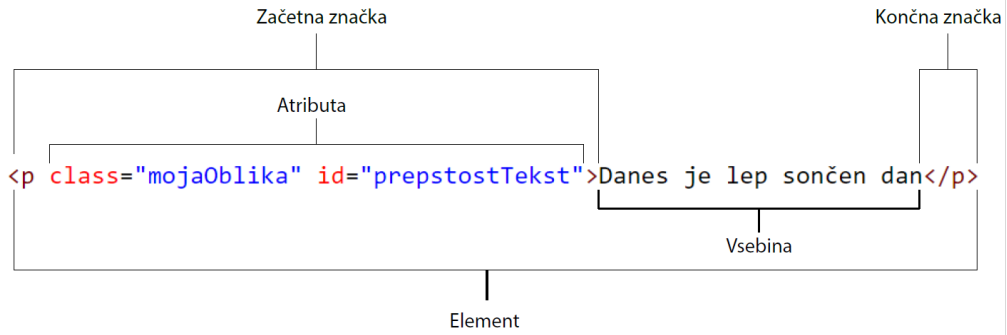
3.1.1 HTML5

Označevalni jezik HTML (HyperText Markup Language) oziroma njegova najnovejšo verzijo HTML5, lahko opišemo kot osnovni gradnik spletnih strani, saj določa njihovo strukturo in vsebino, omogoča oblikovanje večpredstavnostnih dokumentov, ki nam omogočajo povezave znotraj dokumenta ali med dokumenti. HTML5 je najnovejša različica standarda HTML in je, kot standardiziran označevalni jezik v uporabi od leta 2012 [13].

HTML nam omogoča, da s pomočjo značk, kot so *<head>*, *<title>*, *<body>*, *<header>*, *<footer>*, *<article>*, *<section>*, *<p>*, * *, **,

<*img*>, določimo strukturo spletne strani, definiramo naslov, med sabo ločimo odstavke, v dokument dodajamo slike in videe ter povezave do drugih strani ali drugega dela dokumenta.

Kot prikazuje slika 3.1, vsebini, ki je obdana z značkami, pravimo element. Vsak element ima začetno in končno značko, poznamo pa tudi samostojne značke, ki ne potrebujejo zaključka. Vsakemu elementu lahko priredimo tudi atribute, v katerih lahko podamo razred, ki nam določa način oblikovanja, povezavo, določimo id elementu itd.



Slika 3.1: Primer HTML elementa

3.1.2 CSS3

CSS (Cascading Style Sheets) je jezik za oblikovanje spletnih strani in njihovih elementov v slovenščini, poznan kot kaskadne slogovne predloge. Omogoča nam, da določimo stil posameznim ali več elementom hkrati. Določanje stila zajema preproste operacije, kot je določanje barv, poravnava, zamik, velikost pisave, obrobe, pa tudi bolj zapletene, kjer lahko na spletno stran uvajamo animacije. CSS nam omogoča bolj pregledno kodo in lažje stilsko oblikovanje spletne strani, saj je mogoče vso oblikovanje implementirati tudi v atributih HTML značk, vendar je zaradi po navadi obsežne kode, ki je potrebna za oblikovanje, in dejstva, da je potrebno posebej definirati stil v atributih vsa-

kega elementa, veliko bolj priročno imeti poseben dokument z oblikovanjem, ki nam omogoča, da isto stil priredimo več elementom [15].

3.1.3 JavaScript in TypeScript

Če želimo razumeti programski jezik TypeScript, moramo najprej razumeti, zakaj se uporablja programski jezik JavaScript, saj je TypeScript samo nadgradnja le-tega. JavaScript je objektno orientiran skriptni jezik, ki se uporablja predvsem za spletno programiranje tako na strežnikih kot na odjemalčevi strani. Poleg osnovnega nabora operatorjev, operacij in objektov, ki jih programski jeziki po navadi ponujajo, JavaScript na odjemalčevi strani omogoča tudi manipuliranje z objektnim modelom dokumenta (ang. DOM) in brskalnikom. To nam omogoča dinamično spreminjanje vsebine spletnih strani, animacije pa tudi implementacijo poslovne in aplikacijske logike na odjemalcu ter asinhrono komunikacijo s strežnikom ali drugimi programskimi vmesniki. Če pa JavaScript uporabljamo na strežniku, ta ponuja funkcionalnosti, ki so pomembne za strežbo, kot je na primer dostop do podatkovne baze [17].

JavaScript se zgleduje po programskem jeziku Java, saj ima podobno sintakso in poimenovanja, vendar je JavaScript veliko manj strog jezik. Ni nam potrebno deklarirati vseh spremenljivk, metod in razredov, ni nam potrebno skrbeti, ali so metode javne ali zasebne, ni nam potrebno definirati kakšnega tipa so spremenljivke itd.

JavaScript v našo spletno stran oziroma v HTML dokument vključimo z značko <script>. V enem dokumentu lahko deklariramo več značk <script> in tako z enim HTML dokumentom povežemo več JavaScript dokumentov.

TypeScript je, kot smo že omenili, samo nadgradnja JavaScripta, ki je namenjen lažjemu razvoju in je bolj prijazen do razvijalcev. Ko pa programsko kodo prevedemo, se ta prevede v čisto JavaScript kodo.

3.2 Angular

Angular je JavaScript ogrodje, ki nam omogoča izgradnjo enostranskih aplikacij (ang. single-page applications). Enostranske aplikacije so spletne aplikacije, pri katerih se nalaganje osnovnega HTML dokumenta izvede samo enkrat, nato pa se prehajanje med stranmi samo simulira. Enostranske spletne aplikacije so torej sestavljene iz enega HTML dokumenta in nekaj JavaScript dokumentov, ki skrbijo za vso logiko, ki je potrebna za simulacijo navigacije med stranmi, prikazovanje različnih delov HTML dokumenta, manipuliranje z objektnim modelom dokumenta in poslovno ter aplikacijsko logiko.

3.2.1 Prednosti enostranskih spletnih aplikacij

Prednosti enostranskih spletnih aplikacij so večje zmogljivosti in večja odzivnost spletne aplikacije, saj ko se aplikacija enkrat naloži, ni več potrebe po prenašanju dokumentov s strežnika ob navigaciji na podstran. Enostranske aplikacije razbremenijo strežnike, saj prihaja na strežnik manj zahtev, logični del, ki skrbi za odzivnost in izgled strani, pa je preseljen na odjemalčevo stran. Enostranske aplikacije poenostavijo izgradnjo odzivnih spletnih aplikacij. Angular je eno izmed ogrodij, ki poenostavi izgradnjo enostranskih spletnih aplikacij, zato smo se odločili, da spletno aplikacijo, ki bo demonstrirala delovanje spletne aplikacije, ki za delovanje ne potrebuje interneta, naredimo kot enostransko aplikacijo s pomočjo ogrodja Angular.

3.2.2 Zgradba Angular projekta

V ogrodju Angular spletno aplikacijo razdelimo na več logičnih delov, imenovanih komponente. Komponente upravljajo z delom zaslona, imenovanem pogled (ang. view). Komponente so poljubno veliki ali majhni deli naše aplikacije, lahko je to na primer celotna podstran ali pa samo en element na strani. Vsaka komponenta mora obvezno vsebovati HTML dokument, imenovan predloga (ang. template), CSS dokument imenovan stil in TypeScript dokument, ki je komponenta. Pri tem moramo biti pozorni, da so to samo

delni dokumenti, ki bodo ob prevajanju združeni v skupen HTML in CSS dokument, ter nekaj JavaScript dokumentov (TypeScript se ob prevajanju prevede v JavaScript). Komponente omogočajo večjo preglednost kode in logično ločitev spletne aplikacije na dele, ki jo sestavljajo.

Poleg komponent poznamo tudi storitve. V storitvah imamo implementirano aplikacijsko logiko, kot je pridobivanje oziroma pošiljanje podatkov preko programskega vmesnika REST (ang. Representational State Transfer), torej aplikacijsko logiko, ki je neodvisna od komponent in jo uporablja več komponent.

Vse skupaj pa združujejo moduli, ki povežejo skupaj komponente in storitve. V njih so deklarirane tudi vse knjižnice, ki smo jih uporabili, od tega so nekatere obvezne druge pa opcijske. Vsaka aplikacija ima vsaj en korenski modul, pri večjih projektih pa je lahko število modulov večje. Na podlagi informacij v modulih zna Angular ob prevajanju združiti vso kodo, ki smo jo napisali v komponentah in storitvah, v en HTML dokument in nekaj JavaScript dokumentov, ki nato tvorijo enostransko spletno aplikacijo. V produkciji se uporablja prevajanje vnaprej (ang. ahead of time), kar pomeni, da aplikacijo prevedemo preden jo namestimo na strežnik. S tem pridobimo na času izvajanja, sama aplikacija pa je tudi manjša, saj ne potrebuje prevajalnika.

V diplomski nalogi smo uporabili Angular verzije 4, ki je kompatibilen z verzijo 2, s prejšnjimi verzijami pa ne. Zaradi uradnega poimenovanja, ki predvideva, da se tako Angular2 kot Angular4 poimenujeta Angular, se v diplomi vedno uporablja samo izraz Angular.

Poglavje 4

Razvoj spletne aplikacije

Do sedaj smo opisovali tehnologije, ki omogočajo uporabo spletnih aplikacij tudi v nepovezanem načinu, in tehnologije, ki smo jih uporabili za samo izgradnjo aplikacije. Na tem mestu pa je čas, da bralcu predstavimo uporabnost naše spletne aplikacije in njene funkcionalnosti.

Spletna aplikacija Patronažna služba je namenjena patronažnim sestram in jim omogoča vnašanje in pregledovanje meritev vitalnih znakov pacientov. Ker je namen diplomske naloge izdelati spletno aplikacijo, ki deluje brez internetne povezave in nam aplikacija za patronažne sestre služi samo, kot primer aplikacije, kjer bi takšna funkcionalnost prišla prav, ta ponuja zelo omejen nabor funkcionalnosti. Pravzaprav samo dve, vnašanje in pregledovanje meritev vitalnih znakov ter shranjevanje pacientov in njihovih zadnjih meritev v lokalno shrambo. Vendar to zadostuje za namen predstavitve predlogov rešitev, uporabe vseh običajnih funkcionalnosti spletnih aplikacij tudi v nepovezanem načinu. Te so: dostopanje do spletne aplikacije in vseh njenih podstrani, prijava, registracija, prikazovanje, shranjevanje in brisanje tekstovnih, številskih, multimedijskih ali objektnih podatkov ter sinhronizacija teh podatkov s strežnikom.

4.1 Zaledni del

Pri diplomski nalogi smo se osredotočili predvsem na izgradnjo čelnega dela (ang. frontend) spletne aplikacije in ne tudi njenega zalednega dela. Za zaledni del smo uporabili storitvi Firebase, ki nam omogoča shranjevanje in avtentikacijo uporabnikov ter gostovanje spletne aplikacije in HAPI-FHIR, ki nam omogoča shranjevanje meritev vitalnih znakov pacientov.

4.1.1 Firebase

Firebase je ena od Googlovih storitev, ki omogoča izgradnjo spletnih in mobilnih aplikacij brez potrebe po programiranju in implementaciji strežniške logike. Nudi nam storitev podatkovne baze, avtentikacije in avtorizacije uporabnikov, gostovanje spletni aplikacij in še več drugih storitev, ki pa za razvoj naše aplikacije niso pomembne [1].

Storitev Firebase smo uporabili za gostovanje naše spletne aplikacije in za shranjevanje podatkov o uporabnikih in avtentikacijo.

Gostovanje

Za gostovanje spletne aplikacije na storitvi Firebase smo se odločili, ker je storitev brezplačna in preprosta za uporabo, predvsem pa zato, ker omogoča gostovanje preko varne HTTPS povezave, kar je nujno, če želimo uporabljati skripto Service Worker.

Avtentikacija

Firebase podpira prijavo z že obstoječimi računi, ki jih ima uporabnik na Googlu, Facebooku, Twitterju ali pa z uporabniškim imenom in geslom. Mi smo se odločili za slednje.

Uporabnika je potrebno najprej registrirati. To storimo tako, da na strežnik pošljemo elektronski naslov in geslo uporabnika ter njegove osebne podatke, ki jih Firebase shrani. Ko se želi uporabnik nato prijaviti, se prijavi z vnosom elektronskega naslova in gesla, njegovi podatki se pošljejo na

strežnik Firebase, ta preveri podatke in če so pravilni odgovori z žetonom (ang. token). Žeton se nato uporablja za vsako naslednjo avtorizacijo uporabnika pri dostopanju do podstrani spletne aplikacije ali do podatkov v podatkovni bazi, veljavnost žetona pa je časovno omejena.

4.1.2 HAPI-FHIR

Odprtokodno storitev HAPI-FHIR, ki je javanska implementacija standarda FHIR (ang. Fast Healthcare Interoperability Resources) oziroma hitra interoperabilnost virov v zdravstvu, kot bi to lahko prevedli v slovenščino. FHIR je standard na področju izmenjave zdravstvenih podatkov. Javni testni strežnik HAPI-FHIR nem je omogočil, da smo lahko shranjevali meritve vitalnih znakov in podatke o pacientih [18].

FHIR različne tipe podatkov, ki jih lahko shranimo na strežnik imenuje viri (ang. resources). Pri diplomski nalogi samo potrebovali samo vira meritev (ang. observation) in pacient (ang. patient). Zato se bomo tukaj omejili samo na predstavitev teh dveh virov.

Meritve

Kot je prikazano v programski kodi 4.1, **meritev** sestavljajo:

- metapodatki, v katerih so shranjeni podatki o verziji in zadnji posodobitvi,
- identifikator skupine meritev, nam služi za identifikacijo virov, pri čemer ima lahko več virov isti identifikator. V našem primeru je to koristno, da lahko na javnem testnem strežniku poiščemo vse meritve, ki so naše,
- kategorija, ki je sestavljena iz podatka o vrsti kodiranja in kater sistem kodiranja uporablja,
- koda oziroma merske enote, ki vsebuje mersko enoto in sistem kodiranja, ki ga uporabljamo,

- id pacienta,
- ter vrednost meritve.


```
{
  "resourceType": "Observation",
  "id": "158407",
  "meta": {
    "versionId": "1",
    "lastUpdated": "2017-06-28T10:40:20.496-04:00"
  },
  "identifier": [
    {
      "value": "patronaza1"
    }
  ],
  "category": [
    {
      "coding": [
        {
          "system": "http://hl7.org/fhir/observation-category",
          "code": "vital-signs",
          "display": "Vital Signs"
        }
      ],
      "text": "body weight"
    }
  ],
  "code": {
    "coding": [
      {
        "system": "http://loinc.org",
        "code": "3141-9",
        "display": "body weight"
      }
    ],
    "text": "body weight"
  },
```

```
35   "subject": {
36     "reference": "Patient/144532"
37   },
38   "valueQuantity": {
39     "value": 72,
40     "unit": "kg",
41     "system": "http://unitsofmeasure.org",
42     "code": "kg"
43   }
44 }
```

Programska koda 4.1: Primer zapisa meritve na strežniku HAPI-FHIR

Pacient

Kot je prikazano v programski kodi 4.2, je **pacient**, sestavljen iz naslednjih podatkov:

- metapodatke o verziji in zadnji posodobitvi,
- identifikator skupine pacientov, ki ima enako vlogo kot pri meritvah,
- ime in priimek, spol ter datum rojstva
- naslov, ki je sestavljen iz ulica, poštne številke, kraja in države,
- ter povezave do fotografije.

```
1  {
2    "resourceType": "Patient",
3    "id": "144533",
4    "meta": {
5      "versionId": "1",
6      "lastUpdated": "2017-06-12T16:26:55.410-04:00"
7    },
8    "text": {
9      "status": "generated",
10   },
11   "identifier": [
```

```
12      {
13        "value": "patronaza1"
14      }
15    ],
16    "name": [
17      {
18        "family": "Novak",
19        "given": [
20          "Ana"
21        ]
22      }
23    ],
24    "gender": "female",
25    "birthDate": "1991-11-06",
26    "address": [
27      {
28        "use": "home",
29        "line": [
30          "Slovenska cesta 34"
31        ],
32        "city": "Ljubljana",
33        "postalCode": "1000",
34        "country": "Slovenia"
35      }
36    ],
37    "photo": [
38      {
39        "contentType": "image/jpeg",
40        "url": "https://cdn.pixabay.com/photo/2016/06/21/23/05/
      girl-1472185_960_720.jpg"
41      }
42    ]
43  }
```

Programska koda 4.2: Primer zapisa pacienta na strežniku HAPI-FHIR

4.2 Uporabnikova pot

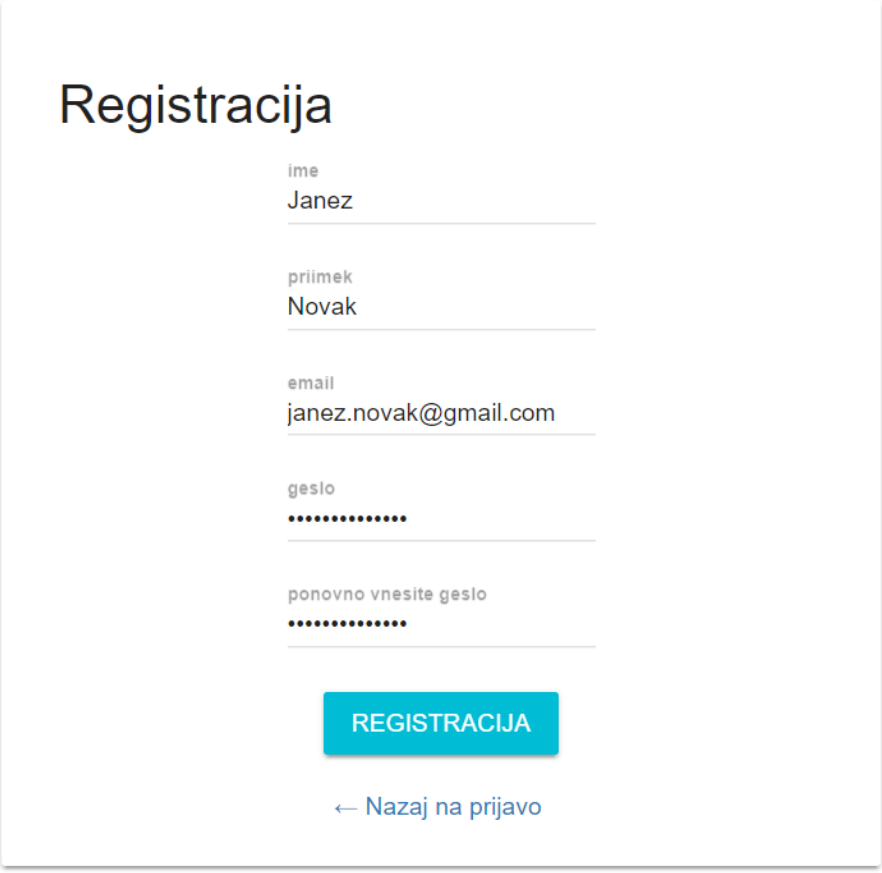
4.2.1 Prijava in registracija

Uporabnik se takrat, ko je povezan v internet prijavi v aplikacijo, z uporabniškim imenom, za kar služi elektronski naslov, in geslom (glej sliko 4.1). Če uporabnik še ni registriran, lahko sledi povezavi do registracije, kjer se z vpisom osebnih podatkov lahko registrira (glej sliko 4.2).

Slika 4.1: Prijava v aplikacijo

4.2.2 Pregled meritev

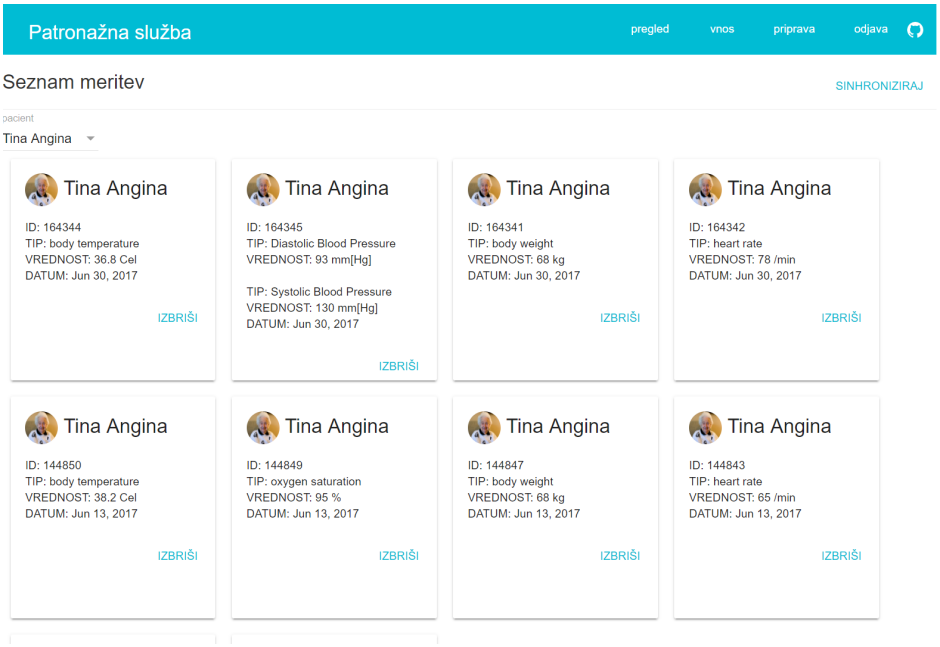
Ko je uporabnik prijavljen, je preusmerjen na vstopno stran, na kateri lahko uporabnik pregleduje meritve vitalnih znakov. To stori tako, da izbere pacienta, za katerega bi želel pridobiti shranjene meritve, ki se mu nato v obliki kartic prikažejo urejene po datumu od najnovejše merivev do najstarejše, po deset meritev na stran. Pri vsaki meritvi imamo tudi možnost, da jo izbrišemo (glej sliko 4.3).



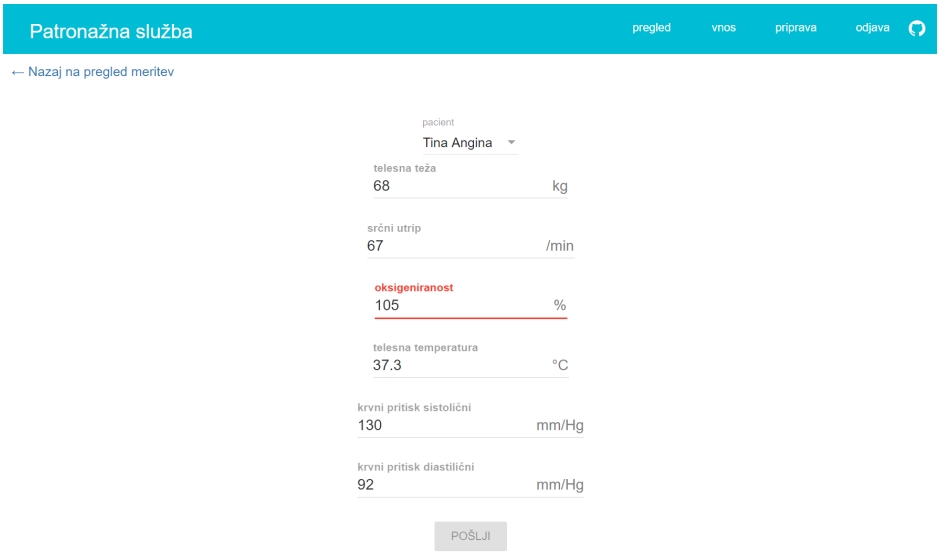
Slika 4.2: Registracija

4.2.3 Vnos meritev

Če želi uporabnik vnesti novo meritev, lahko v meniju izbere *vnos*, ki ga preusmeri na stran za vnašanje novih meritev vitalnih znakov. Če je pacient pri pregledu meritev že izbran, se ta pri vnosu avtomatsko nastavi, če ne, ga uporabnik izbere iz spustnega seznama. Uporabnik lahko nato vnese eno ali več meritev osnovnih vitalnih funkcij, pri čemer sepreveri, ali je vnesel veljavno in smiselno vrednost. Z gumbom *Pošlji* se meritve pošljejo na strežnik, oziroma shranijo v lokalno shrambo, če smo v nepovezanem načinu, o čemer je uporabnik tudi obveščen (glej sliko 4.4).



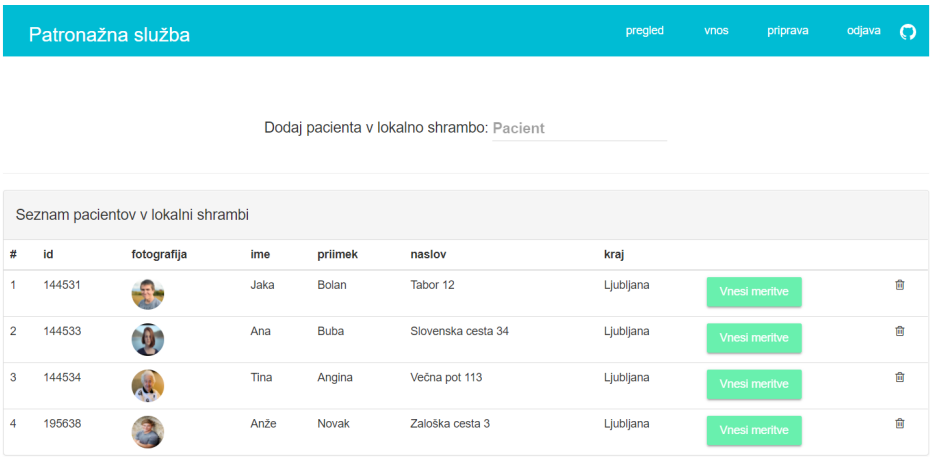
Slika 4.3: Primer pregleda meritev za enega od pacientov



Slika 4.4: Primer vnosa meritev in napake

4.2.4 Priprava plana obiskov

Uporabnik si zjutraj, preden gre po obiskih, naredi seznam pacientov, ki jih bo obiskal in paciente ter njihove osebne podatke in najnovejše meritve shrani v lokalno shrambo. Uporabnik lahko to stori s klikom na zavihek *pripravi*, kjer s spustnega seznama izbere pacienta ali pa ga poišče glede na priimek ali ime ter ga s klikom doda na seznam pacientov, ki se shranijo v lokalni shrambi. Tukaj lahko uporabnik pregleduje podatke o pacientu in jih briše s seznama (glej sliko 4.5).



Slika 4.5: Seznam pacientov pripravljenih za obisk in polje za dodajanje novih

4.2.5 Uporaba v nepovezanem načinu

Ko uporabnik izgubi povezavo do interneta in preide v nepovezan način, se mu ob dostopu do aplikacije ni potrebno prijaviti v aplikacijo, ampak ga takoj preusmerimo na predhodno pripravljen seznam pacientov, ki jih mora obiskati ta dan in so njihovi podatki ter najnovejše meritve shranjene v lokalni shrambi. Iz seznama lahko izbere pacienta za katerega želi vnesti meritve, meritve se vnašajo na enak način, kot pri povezanem načinu. Uporabnik lahko tudi enako, kot v povezanem načinu, na zavihku *pregled*, pregleduje

meritve pacientov. Le da ima sedaj na voljo le paciente, ki so lokalno shranjeni. Od teh pacientov pa lahko prav tako pregleduje le lokalno shranjene meritve, ki so najnovejše meritve vsakega tipa in meritve, ki jih je uporabnik vnesel v nepovezanem načinu. V nepovezanem načinu uporabnik nima možnosti priprave seznama pacientov, ki se shranijo v lokalno shrambo.

4.2.6 Sinhronizacija

Na strani s pregledom meritev ima uporabnik možnost, da s klikom na gumb *SINHRONIZIRAJ* sinhronizira podatke. V primeru, da je uporabnik spletno aplikacijo nekaj časa uporabljal v nepovezanem načinu in pri tem vnesel nove meritve ali izbrisal katero od meritev, se bodo ob sinhronizaciji (seveda mora biti uporabnik v tem primeru povezan v internet) na strežnik poslale nove vnesene meritve in zahteva za izbris meritev. Ob tem pa se bodo izbrisali tudi vsi lokalno shranjeni podatki o pacientih in njihovih meritvah.

Poudarimo še, da uporabnik ne zazna prehoda iz povezanega v nepovezan način in lahko aplikacijo nemoteno uporablja, edino, kar zahtevamo od njega, je, da v primeru, da je spreminjal, vnašal, oziroma brisal meritve jih mora ročno sinhronizirati s strežnikom, ko je spet povezan v internet.

4.3 Implementacija logike za delovanje v nepovezanem načinu

4.3.1 Prvi dostop do strani

Ko uporabnik prvič dostopa do spletne aplikacije na izbrani napravi, se po tem, ko se je prenesla HTML datoteka, med drugimi skriptami prenese in namesti tudi skripta Service Worker. Ob namestitvi v lokalno shrambo prenese vse potrebne spletne vire, ki so potrebni, da naša spletna aplikacija deluje. Od sedaj naprej sicer do naše spletne aplikacije lahko dostopamo v nepovezanem načinu, ni pa se še mogoče pregledovati meritve in paciente v

nepovezanem načinu, kar bo mogoče šele po prvi prijavi in pripravi plana lokalno shranjenih pacientov.

Ob prvem dostopu do aplikacije pa se pri uporabniku ustvari tudi podatkovna baza IndexedDB, ki se uporabljala za shranjevanje vseh podatkov, potrebnih za uporabo aplikacije v nepovezanem načinu. Ob postavitvi podatkovne baze se kreirajo naslednje shrambe:

- **patients** - Shranjuje podatke o pacientih. Shramba vsebuje id pacienta in JSON objekt z vsemi podatki pacienta, kot je prikazan v programski kodi 4.2.
- **observations** - Shranjuje podatke o meritvah vitalnih znakov. Shramba vsebuje id meritve, id pacienta in celoten JSON objekt meritve, kot je prikazan v programski kodi 4.1.
- **deleteQueue** - Shramba vsebuje id meritev, ki jih je uporabnik izbrisal, ko je bil v nepovezanem načinu.
- **observationQueue** - Shramba, ki vsebuje tip, podtip in vrednost meritev, datum opravljene meritve ter paciente, na katerega so vezane meritve, ki jih je uporabnik vnesel v nepovezanem načinu.

4.3.2 Prijava in registracija

Naslednji izziv, ki ga je bilo treba rešiti, je prijava in registracija v sistem. Kot smo že omenili in opisali v podpoglavju 4.1.1, smo za strežniško logiko registracije in prijave uporabili storitev Firebase.

Če uporabnik še ni registriran, se lahko registrira z elektronskim naslovom in geslom, ta se pošljeta na strežnik Firebase, ki ustvari novega uporabnika.

Če ima uporabnik dostop do internetne povezave, prijava poteka tako, da se elektronski naslov in geslo uporabnika pošljeta na strežnik Firebase. Ta preveri, ali so podatki pravilni, in če so, vrne žeton, ki pomeni, da so podatki pravilni, torej lahko uporabniku dovolimo dostop do spletne aplikacije,

hkrati pa se ta žeton lahko uporablja za avtorizacijo vseh nadaljnjih dostopov do podstrani ali podatkov v podatkovni bazi. Žetona v naši aplikaciji za nadaljnjo avtorizacijo nismo uporabili, saj imamo vse podatke shranjene na strežniku HAPI-FHIR.

Če uporabnik uporablja aplikacijo v nepovezanem načinu, možnosti za prijavo nima in aplikacijo vedno uporablja neprijavljen, razlog za to je predvsem varnost, ki jo bomo podrobneje razložili v podpoglavju 4.3.6.

4.3.3 Pregled meritev

Ko je uporabnik enkrat prijavljen, lahko pregleduje vnesene meritve vitalnih znakov, glede na izbranega pacienta. Če je uporabnik povezan v internet, se seznam pacientov prenese s strežnika HAPI-FHIR, med podatki je tudi URL prikazne slike pacienta, ki se prenese ločeno. Ko uporabnik izbere pacienta, za katerega želi pregled meritev, se na strežnik HAPI-FHIR pošlje zahteva po meritvah. Zahteva je sestavljena iz našega identifikatorja, ki označuje meritve, ki jih je vnesel katerikoli od naših uporabnikov. To je potrebno zaradi javne narave strežnika HAPI-FHIR. V zahtevi pa se pošlje še id pacienta, ki strežniku pove po meritvah katerega pacienta poizvedujemo in odmik ter število meritev, ki jih želimo prejeti. S tem omogočimo paginacijo meritev za lažji pregled.

V primeru, da je uporabnik v nepovezanem načinu, vse podatke o meritvah in pacientih dobimo iz lokalne shrambe, katere smo s strežnika HAPI-FHIR prenesli ob pripravi plana obiskov pacientov. Meritve lahko v nepovezanem načinu pregledujemo le za paciente, ki smo jih predhodno shranili, zanje pa imamo shranjene samo eno, najnovejšo meritev vsakega tipa. Razlog za samo eno shranjeno meritev vsakega tipa, je v varovanju osebnih podatkov.

4.3.4 Vnašanje in brisanje meritev

Tudi tukaj ločimo delovanje v povezanem oziroma nepovezanem načinu. Ne glede na to ali smo v povezanem ali nepovezanem načinu se nove meritve, ki jih je uporabnik vnesel, najprej shranijo v shrambo *observationQueue* in šele na to poskusimo meritve poslati na strežnik. Če s strežnika dobimo odgovor, da smo vse meritve uspešno poslali, meritve iz lokalne shrambe pobrišemo. V primeru, da s strežnika dobimo odgovor, da je prišlo do napake ali pa od brskalnika, da meritev ni bilo mogoče poslati, kar pomeni, da smo v nepovezanem načinu, meritve ostanejo v shrambi *observationQueue* in se bodo na strežnik poslale, ko bo uporabnik aplikacijo sinhroniziral s strežnikom. Preden meritve pošljemo na strežnik jih je potrebno formatirati, kot to zahteva standard FHIR, prikazano v programski kodi 4.1.

Po istem načelu deluje tudi brisanje meritev. Id izbrisane meritve se najprej shrani v shrambo *deleteQueue* nato pa se pošlje zahteva na strežnik FIRE-BASE za brisanje meritve. Če ta uspe, se meritev izbriše tudi v shrambi *deleteQueue*. Če pa pride do napake, meritev ostane v shrambi in se ponovno poskusi izbrisati meritev kasneje, ko imamo internetno povezavo. Hkrati pa se meritev izbriše tudi v shrambi *observations*.

4.3.5 Sinhronizacija

Na strani s pregledom meritev imamo tudi gumb *SINHRONIZIRAJ*. To uporabniku omogoči, da lahko po tem, ko je v nepovezanem načinu vnašal in brisal meritve, te sinhronizira s strežnikom, ko ima internetno povezavo spet vzpostavljeno. Ob pritisku na gumb aplikacija poskuša vse meritve, ki so v shrambi *observationQueue* poslati na strežnik in vse meritve, ki so v *deleteQueue* izbrisati na strežniku. Če se operaciji uspešno izvršita, se meritve in vsi podatki o pacientih pobrišejo iz lokalne shrambe.

4.3.6 Varnost

Pri izgradnji aplikacije nismo dajali velikega poudarka varnosti, ki bi sicer v aplikaciji za zdravstvene namene bila zelo pomembna. Zato je prav, da opozorimo na varnostne pomanjkljivosti v naši aplikaciji.

Razlog zakaj se uporabnik lahko prijavi samo v povezanem načinu je ta, da če bi hoteli prijavo v aplikacijo omogočiti tudi v nepovezanem načinu, bi morali imeti lokalno shranjene podatke vseh uporabnikov. To bi bil velik varnostni problem, saj bi lahko vsi uporabniki videli podatke ostalih uporabnikov in njihova gesla. Gesla bi sicer lahko shranili v šifrirani obliki, oziroma bi to bilo vsekakor nujno, vendar v vsakem primeru ni varno lokalno shranjevati gesel, tako da imajo vsi ostali uporabniki dostop do njih. S omejitvijo, da se uporabnik prijavlja samo v povezanem načinu, tako zagotovimo, da do podatkov osatlih uporabnikov ne more dostopati nihče, do podatkov o meritvah pacientov pa lahko dostopajo samo avtorizirani uporabniki.

Bolje moramo pojasniti tudi našo izbiro shranjevanja samo ene, najnovejše meritve vsakega tipa. V nepovezanem načinu, se glede varnosti zanašamo izključno na geslo, ki varuje napravo (geslo, ki odklepa mobilni telefon ali računalnik). Če ima uporabnik dostop do naprave, ima prost dostop tudi do aplikacije in njenih podatkov. Z omejitvijo količine lokalno shranjenih podatkov tako omejimo škodo, ki bi nastala, če bi nekdo udrl v napravo. Poudariti pa je potrebno tudi to, da nas pred krajo podatkov iz lokalne shrambe ne bi varovala niti avtentikacija v aplikacijo, saj lahko do podatkov v lokalni shrambi preko orodij brskalnika dostopamo ne, da smo prijavljeni.

Še dodatno bi podatke o pacientih in njihovih meritvah lahko zavarovali tako, da bi namesto pravih imen pacientov uporabljali šifre ali namišljena uporabniška imena, s čimer bi zagotovili, da če tudi bi nepooblaščena oseba prišla do podatkov o meritvah, teh ne bi znala povezati z osebo kateri pripadajo.

4.4 Arhitektura aplikacije

4.4.1 Vzpostavitev ogrodja Angular

Kot smo že omenili, smo aplikacijo razvili s pomočjo ogrodja Angular. Najprej ga je bilo potrebno namestiti. Ob namestitvi smo ustvarili nov projekt *diploma*, Angular pa je poskrbel za vzpostavitev osnovnega ogrodja. Osnovno ogrodje poleg dokumentov z okoljskimi spremenljivkami in obveznih knjižnic, vsebuje korensko komponento in korenski modul, v katerem smo navedli vse odvisnosti od komponente in storitve.

4.4.2 Service Worker

Skripto Service Worker, ki jo uporabljamo za shranjevanje virov spletnih strani v lokalno shrambo in prestrezanje zahtev, ter odgovarjanje nanje, najprej registriramo v datoteki *indexed.html* (glej programsko kodo 4.3), sama skripta pa je shranjena v datoteki *service-worker.js*. V skripti najprej navedemo vse spletne vire, ki jih želimo shraniti, nato pa poslušamo za dogodka *install* in *fetch*. Ob dogodku *install* v lokalno shrambo shranimo vse spletne vire, ob dogodku *fetch*, pa preverimo ali imamo v lokalni shrambi vir, ki ga zahtevamo, če obstaja ga vrnemo, če ne ga poskušamo pridobiti iz interneta (glej programsko kodo 4.4).

```
<script>
    if ('serviceWorker' in navigator) {
      navigator.serviceWorker.register('/service-worker.js').
    then(function(registration) {
        console.log('Service Worker registered');
      }).catch(function(err) {
        console.log('Service Worker registration failed: ', err)
    ;
      });
    }
  </script>
```

Programska koda 4.3: Registracija skripte Service Worker.

```
const CACHE = 'my-cache-v2';
const urlsToCache = [
  '/',
  'index.html',
  'inline.bundle.js',
  'polyfills.bundle.js',
  'styles.bundle.js',
  'vendor.bundle.js',
  'main.bundle.js'
];

self.addEventListener('install', function (event) {

  event.waitUntil (
    caches.open(CACHE).then(function (cache) {
      return cache.addAll(urlsToCache);
    })
  )
});

self.addEventListener('fetch', function (event) {
  event.respondWith(
    caches.match(event.request).then(function (response) {
      if( response)
        return response;
      return fetch(event.request);
    })
  )
})
```

Programska koda 4.4: Skripta Service Worker.

4.4.3 Storitve

Za potrebe izgradnje aplikacije smo ustvarili naslednje storitve:

- **indexeddb service** - Storitev vsebuje funkcijo za inicializacijo lokalne podatkovne baze IndexedDB (glej programsko kodo 4.5) in nje-

nih shramb. Funkcije za dodajanje, pridobivane in brisanje podatkov iz shramb.

Storitev vsebuje funkcije za dodajanje meritev v shrambo *obseravtionQueue*, pridobivanje vseh meritev in glede na id uporabnika ter brisanje vseh meritev v tej shrambi. Storitev vsebuje tudi funkcije za dodajanje meritev v shrambo *observations* (glej programsko kodo 4.6), pridobivanje meritev iz te shrambi glede na id meritve in pridobivanje vseh meritev, brisanje vseh meritev, brisanje meritev glede na id meritve in glede na id uporabnika. Storitev vsebuje tudi funkcije za dodajanje, brisanje in pridobivanje vseh meritev v shrambo *deleteQueue* in funkcije za dodajanje pacientov v shrambo *patients*, pridobivanje vseh pacientov in glede na id pacienta, ter brisanje vseh paientov ali pa glede na njihov id.

```
1  initializeDB() {
2      this.db = new AngularIndexedDB();
3      this.db.createStore(this.DB_VERSION, (evt) => {
4
5          let objectStore = evt.currentTarget.result.
       createObjectStore(
6              'observations', {keyPath: 'id', autoIncrement: true
       });
7
8          objectStore.createIndex('id', 'id', {unique: true});
9          objectStore.createIndex('patientId', 'patientId', {
       unique: false});
10         objectStore.createIndex('observation', 'observation',
       {unique: false});
11
12         objectStore = evt.currentTarget.result.
       createObjectStore(
13             'observationQueue', {keyPath: 'id', autoIncrement:
       true});
14
15         objectStore.createIndex('type', 'type', {unique: false
       });
```

```
16        objectStore.createIndex('value', 'value', {unique:
    false});
17        objectStore.createIndex('subtype', 'subtype', {unique:
     false});
18        objectStore.createIndex('patient', 'patient', {unique:
     false});
19        objectStore.createIndex('date', 'date', {unique: false
    });
20
21        objectStore = evt.currentTarget.result.
    createObjectStore(
22            'deleteQueue', {keyPath: 'id', autoIncrement: true})
    ;
23
24        objectStore.createIndex('id', 'id', {unique: true});
25
26        objectStore = evt.currentTarget.result.
    createObjectStore(
27            'patients', {keyPath: 'id', autoIncrement: true});
28
29        objectStore.createIndex('id', 'id', {unique: true});
30        objectStore.createIndex('patient', 'patient', {unique:
     false});
31      });
32    }
```

Programska koda 4.5: Programska koda za inicializacijo podatkovne baze IndexedDB in njenih shramb.

```
1 addObservation(observation: any, id: number): number {
2     this.db.add('observations', {
3       observation: observation,
4       id: id,
5       patientId: observation.resource.subject.reference.
    split('/')[1]
6     }).then((_observation) => {
7       console.log('uspesno dodana meritev ' + _observation);
8       return 1;
```

```
9     }, (error) => {
10      console.log('napaka pri dodajanju meritve ' + error);
11      return 0;
12    });
13    return 0;
14  }
```

Programska koda 4.6: Programska koda za dodajanje meritev v shrambo observations.

- **observation service** - Storitev, ki vsebuje funkcije za komunikacijo s strežnikom HAPI-FHIR. Vsebuje funkcijo za pridobivanje meritev s strežnika, ki ji podamo identifikacijo, odmik in število meritev ter id pacienta, ki jih želimo prejeti. Funkcijo, ki nam s strežnika vrne podatke o pacientu, katerega id smo podali funkciji, ter funkciji za pošiljanje in brisanje meritev na strežniku.

- **user service** - Storitev vsebuje funkcije potrebne za avtentikacijo uporabnika na strežniku Firebase. Storitev vsebuje funkcijo za registracijo uporabnika, ki ji podamo elektronski naslov, ime in priimek, ter geslo uporabnika, ta pa te podatke posreduje Firebase strežniku. Vsebuje funkcijo za prijavo, ki ji podamo elektronski naslov in geslo uporabnika, ki jih posreduje Firebase strežniku, ta preveri pravilnost podatkov in, če so pravilni odgovori z žetonom, ki se shrani v lokalno shrambo. Storitev vsebuje tudi funkciji za odjavo in preverjanje, ali je uporabnik avtenticiran, kar se preveri tako, da se pogleda, ali obstaja veljaven žeton.

- **auth-guard service** - Storitev, ki v ogrodju Angular služi za preverjanje ali je uporabniku dovoljen dostop do podstrani aplikacije. V naši aplikaciji preverjamo samo, ali je uporabnik prijavljen. V tem primeru ima dostop do vseh podstrani. Neprijavljen uporabnik pa samo do strani za prijavo in registracijo.

- **AngularIndexedDB service** - Storitev, ki je v našem projektu samo

zato, ker je prišlo pri nameščanju knjižnice za lažjo komunikacijo s programskim vmesnikom IndexedDB do težav. Kodo knjižnice smo prekopirali in jo v svoj projekt umestili kot storitev in tako dosegli pravilno delovanje.

4.4.4 Predloga meritev

Ker je strežnik HAPI-FHIR javanska implementacija standarda FHIR, ogrodje Angular pa uporablja programski jezik JavaScript oziroma TypeScript, smo morali razred za oblikovanje objektov meritev spisati sami. Razred vsebuje funkcije za generiranje objekta meritev, ki ustreza standardom FHIR, pri tem pa v naši aplikaciji zaradi lažje implementacije ločimo preproste meritve in strukturirano meritev. Preproste meritve so meritve telesne teže, srčnega utripa, nasičenosti krvi s kisikom in telesna temperatura. Kot strukturirano meritev pa definiramo krvni tlak, ki je sestavljen iz sistoličnega in diastoličnega krvnega tlaka. Krvni tlak obravnavamo ločeno, ker imamo v eni meritvi združeni dve vrednosti in je oblika objekta drugačna.

4.4.5 Komponente

Logično strukturo aplikacije smo razdelili na naslednje komponente:

- **login component** - Komponenta, ki upravlja s pogledom prijave in vsebuje HTML dokument z opisom postavitve strani in TypeScript dokument z logiko, ki iz obrazca (ang. form) pridobi elektronski naslov in geslo uporabnika ter ju posreduje storitvi *user service*, ki poskrbi za dejansko prijavo uporabnika.

- **registration component** - Komponenta vsebuje HTML dokument z opisom postavitve strani za registracijo in logiko, ki iz obrazca za registracijo pridobi podatke o uporabniku, ime, priimek, elektronski naslov in geslo. Preden se kliče funkcija za registracijo, se preveri, ali se vneseni gesli ujemata. Če so vsi podatki veljavni, funkcija za

registracijo kliče funkcijo za registracijo v storitvi *user service*, ki izvrši dejansko registracijo uporabnika na strežniku Firebase.

- **observation-input component** - Komponenta vsebuje HTML dokument z opisom postavitve strani za vnašanje meritev in aplikacijo logiko. V aplikacijski logiki imamo definirano vsebino obrazca, torej katere vrednosti uporabnik lahko vnaša, prav tako pa imamo definirane funkcije za validacijo vnosov vrednosti meritev, ki morajo ustrezati smiselnim intervalom.

  Komponenta vsebuje tudi logiko, potrebno za pošiljanje meritev in shranjevanje teh v lokalno shrambo (glej programsko kodo 4.7). Funkcija najprej kliče storitev *indexeddb service*, da meritve shrani v shrambo *observationQueue*, nato kliče storitev *observation service*, preko katere poskuša poslati meritve na strežnik, če to uspe spet s klicem storitve *indexeddb service* izbrišemo meritve iz shrambe *deleteQueue*.

  Komponenta vsebuje tudi funkcijo za pridobivanje pacientov iz strežnika s klicem storitve *observation service* ali iz lokalne podatkovne baze s klicem storitve *indexeddb service*. Seznam pacientov potrebujemo, da lahko uporabniku prikažemo spustni seznam vseh pacientov za katere lahko vnese meritve.

```
1     this.indexedDB.getAllObservationsQueue().then((
    observations) => {
2       if (observations.length > 0) {
3         for (const el of observations) {
4           const entry: any = {};
5           entry.request = this.request;
6           observation = new Observation();
7           if (el.patient) {
8             // zgeneriramo objekt meritve
9             entry.resource = (observation.createObservable
    (el.value, el.type, el.subtype, el.patient));
10          }
11          if (entry.resource !== null) {
12            this.bundle.entry.push(entry);
```

```
13            }
14          }
15          // Meritve poskusimo poslati na streznik, ce uspe
    meritve pobrisemo iz vrste
16          this.observationService.post(this.bundle).
    subscribe(
17            response => {
18              if (response.entry.length === observations.
    length) {
19                this.indexedDB.deleteAllObservationsQueue().
    then();
20                resolve(response.entry.length);
21              } else {
22                reject();
23              }
24            });
25        } else {
26          resolve(0);
27        }
28      });
29    });
30  }
31
```

Programska koda 4.7: Programska koda za pošiljanje meritev na strežnik oz. shranjevanje v lokalno shrambo.

- **observation-list component** - Komponenta vsebuje HTML dokument z opisom postavitve strani za pregled meritev in aplikacijsko logiko. V aplikacijski logiki so funkcije, ki se odzivajo na dogodke. Funkcija *getpatients*, ob inicializaciji komponente pridobiva paciente s klicem storitev iz strežnika ali lokalne podatkovne baze, ter jih shrani v tabelo. Ko uporabnik izbere, za katerega pacienta bi želel pregled meritev, se pokliče funkcija *getObservations* (glej programsko kodo 4.8), ki s klicem storitev pridobi meritve s strežnika ali lokalne shrambe, pri čemer pri odmiku upošteva stran paginacije, na kateri se uporabnik nahaja.

Funkcija shrani seznam meritev v tabelo HTML dokument pa poskrbi, da se te ustrezno prikažejo. Ob izbrisu meritve se pokliče funkcija *onDelete*, ki s kliče storitvi *indexeddb service* in *observation service*, da se izvede postopek brisanja. Ob sinhronizaciji funkcija *onSinc* kliče funkcije za brisanje in pošiljanje meritev.

```
1          // Meritev s streznika ni bilo mogoce pridobiti
   zato poskusimo pridobiti meritve iz IndexedDB-ja
2          () => {
3            const patientId: string = 'Patient/' + this.
   patient.resource.id;
4            this.indexedDB.getObservationsById(patientId).
   then((response: any) => {
5              this.observations = response;
6              if (this.observations.length > 0) {
7                for (const observation of this.observations)
    {
8                  const id = observation.resource.subject.
   reference.substring(8); // iz Patient/123456 se pretvori
    v 123456
9                  // Pridobimo podatke pacienta iz lokalne
   shrambe
10                 this.indexedDB.getPatient(id).then((
   patient) => { observation.patient = patient; });
11               }
12             }
13           });
14           this.indexedDB.getObservationFromQueueByPatient(
   this.patient.resource.id).then((response: any) => {
15             this.queueObservations = response;
16           }); },
17       )}, 10);
18   }
19
```

Programska koda 4.8: Programska koda za pridobivanje meritev iz lokalne shrambe.

- **preparation component** - Komponenta vsebuje HTML dokument z opisom postavitve strani za pripravo seznama pacientov, ki jih bo uporabnik obiskal in prikaza tega seznama ter aplikacijsko logiko potrebno za to. Aplikacijska logika vsebuje funkcijo *filterPatients*, ki omogoča avtomatsko dopolnjevanje pri iskanju uporabnikov, funkcijo *addPatient*, ki v lokalno shrambo shrani vse podatke pacienta in najnovejšo meritev vsakega tipa. Komponenta vsebuje tudi funkcijo *deletepatient*, ki iz seznama in lokalne shrambe izbriše pacienta, ter vse njegove lokalno shranjene meritve.

- **login-navbar component in main-navbar component** - Komponenti, ki vsebujeta HTML dokument s postavitvijo navigacijskega menija in logiko za navigacijo med podstranmi.

- **page-not-found component** - Komponenta, ki vsebuje HTML dokument za prikaz strani, če uporabnik dostopa do podstrani v aplikaciji, ki ne obstaja.

- **app component** - Komponenta, ki jo ogrodje Angular avtomatsko generira in se uporablja samo za združevanje in izhod (ang. outlet) ostalih komponent.

4.4.6 Moduli

Naš projekt vsebuje dva modula. Modul *app module* je korenski modul, ki vsebuje informacije in odvisnosti vseh komponent in storitev ter je potreben pri prevajanju in izgradnji projekta v enostransko aplikacijo. Drug modul pa je modul *app-routing module*, ki je odgovoren za simulacijo navigacije med podstranmi aplikacije. Kot smo razložili v podpoglavju 3.2, so enostranske aplikacije sestavljene iz enega samega HTML dokumenta, zato se mora navigacija med podstranmi simulirati. V modul za navigacijo je vključena storitev *auth-guard service*, ki skrbi, da do podstrani za pregled in vnašanje meritev lahko dostopajo samo prijavljeni uporabniki.

Poglavje 5

Sklepne ugotovitve

V sklopu diplomske naloge smo implementirali spletno aplikacijo za patronažne sestre kot dokaz, da je mogoče s tehnologijami, ki jih ponuja HTML5, ustvariti spletno aplikacijo, ki je uporabna tudi v nepovezanem načinu. Najprej smo pregledali vse tehnologije, ki obstajajo, preučili njihovo delovanje ter se seznanili s pomanjkljivostmi. Sledila je zasnova arhitekture aplikacije, kjer smo se odločili, katere funkcionalnosti želimo, da aplikacija ponuja in se odločili, katere tehnologije bomo uporabili, da bo omogočeno delovanje aplikacije v nepovezanem načinu. Sledila je izgradnja aplikacije, kjer smo jo najprej implementirali kot navadno spletno aplikacijo brez delovanja v nepovezanem načinu in nato dodali vse mehanizme, potrebne za delovanje v nepovezanem načinu.

Z implementacijo spletne aplikacije smo želeli dokazati, da je moč izdelati spletno aplikacijo, ki je uporabna in omogoča večino funkcionalnosti tudi v nepovezanem načinu. Predlagali smo rešitve za implementacijo običajnih funkcionalnosti v nepovezanem načinu, ki jih potrebujejo spletne aplikacije. Najprej seveda sam dostop do aplikacije, nato registracijo in prijavo ter pregled, dodajanje in brisanje podatkov, ne glede na njihovo obliko. Menimo, da nam je uspelo implementirati aplikacijo, ki je popolnoma funkcionalna v nepovezanem načinu, mehanizme, s katerimi smo to dosegli pa lahko brez večjih težav prenesemo na ostale spletne aplikacije.

Zavedamo se, da aplikacija ponuja zelo omejen nabor funkcionalnosti, vendar smo uspeli s tako preprosto aplikacijo zajeti vse glavne izzive, ki jih moramo pri razvoju nepovezanih spletnih aplikacij rešiti. Zavedamo se tudi, da je ostalo še nekaj prostora za izboljšave. Največje pomanjkljivosti, ki smo jih opazili, so predvsem problem varnosti in pa tudi performance aplikacije bi bilo mogoče izboljšati.

Literatura

[1] Firebase documentation. Dosegljivo: https://firebase.google.com/docs/, 2017. [Dostopano: 12.7.2017].

[2] Jake Archibald. Application cache is a douchebag. Dosegljivo: https://alistapart.com/article/application-cache-is-a-douchebag, 2012. [Dostopano: 24.5.2017].

[3] Jake Archibald. The offline cookbook. Dosegljivo: https://jakearchibald.com/2014/offline-cookbook/, 2014. [Dostopano: 7.6.2017].

[4] Eric Bidelman. The basics of web workers. Dosegljivo: https://www.html5rocks.com/en/tutorials/workers/basics/, 2010. [Dostopano: 7.6.2017].

[5] Eric Bidelman. A beginner's guide to using the application cache. Dosegljivo: https://www.html5rocks.com/en/tutorials/appcache/beginner/, 2010. [Dostopano: 24.5.2017].

[6] WHATWG community. Html. Dosegljivo: https://html.spec.whatwg.org/multipage/workers.html#dom-worker-postmessage, 2017.

[7] Rob Crowther, Joe Lennon, Ash Blue, and Greg Wanish. Html5 in action. In *HTML5 in Action*, chapter 5. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2014.

[8] Shwetank Dixit. Web storage: Easier, more powerful client-side data storage. Dosegljivo: https://dev.opera.com/articles/web-storage/, 2013. [Dostopano: 28.6.2017].

[9] reach-star erikadoyle, SphinxKnight. Using the application cache. Dosegljivo: https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache, 2016. [Dostopano: 24. 5. 2017].

[10] Matt Gaunt. Service workers: an introduction. Dosegljivo: https://developers.google.com/web/fundamentals/getting-started/primers/service-workers, 2017. [Dostopano: 7.6.2017].

[11] Ian Hickson. Web sql database. Dosegljivo: https://www.w3.org/TR/webdatabase/#introduction, 2010. [Dostopano: 7.6.2017].

[12] Peter Lubbers, Brian Albers, Ric Smith, and Frank Salim. *Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development*. Apress, Berkely, CA, USA, 1st edition, 2010.

[13] Mozilla Developer Network and individual contributors. Html basics. Dosegljivo: https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics, 2016. [Dostopano: 29.6.2017].

[14] Mozilla Developer Network and individual contributors. Indexeddb api, basic concepts. Dosegljivo: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Basic_Concepts_Behind_IndexedDB, 2016. [Dostopano: 29.6.2017].

[15] Mozilla Developer Network and individual contributors. Css basics. Dosegljivo: https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/CSS_basics, 2017. [Dostopano: 29.6.2017].

[16] Mozilla Developer Network and individual contributors. Indexeddb api. Dosegljivo: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API, 2017. [Dostopano: 29.6.2017].

[17] Mozilla Developer Network and individual contributors. Javascript guide, introduction. Dosegljivo: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction#What_is_JavaScript, 2017. [Dostopano: 30.6.2017].

[18] University Health Network. Hapi-fhir. Dosegljivo: http://hapifhir.io/, 2017. [Dostopano: 30.6.2017].

[19] Mark Pilgrim. Html5: Up and running. In *HTML5: Up and Running*, chapter 7, pages 127–133. O'Reilly Media, Inc., 1st edition, 2010.

[20] Remy Sharp. Introducing web sql databases. Dosegljivo: http://html5doctor.com/introducing-web-sql-databases/, 2010. [Dostopano: 29.5.2017].